LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-522031

# GTI: A Generic Tools Infrastructure for Event Based Tools in Parallel Systems

T. Hilbrich, M. S. Mueller, B. R. de Supinski, M. Schulz, W. E. Nagel

January 5, 2012

International Parallel and Distributed Processing Symposium (IPDPS)
Shanghai, China
May 21, 2012 through May 25, 2012

## Disclaimer

# GTI: A Generic Tools Infrastructure for Event-Based Tools in Parallel Systems

Tobias Hilbrich*, Matthias S. Müller*, Bronis R. de Supinski†, Martin Schulz† and Wolfgang E. Nagel*
*Technische Universität Dresden, ZIH
D-01062 Dresden, Germany,
Email: {tobias.hilbrich, matthias.mueller, wolfgang.nagel}@tu-dresden.de
†Lawrence Livermore National Laboratory
Livermore, CA 94551
Email: {bronis,schulzm}@llnl.gov

***Abstract*—Runtime detection of semantic errors in MPI applications supports efficient and correct large-scale application development. However, current approaches scale to at most one thousand processes and design limitations prevent increased scalability. The need for global knowledge for analyses such as type matching, and deadlock detection presents a major challenge. We present a scalable tool infrastructure – the Generic Tool Infrastructure (GTI) – that we will use to implement MPI runtime error detection tools and that applies to other use cases. GTI supports simple offloading of tool processing onto extra processes or threads and provides a tree based overlay network (TBON) for creating scalable tools that analyze global knowledge. We present its abstractions and code generation facilities that ease many hurdles in tool development, including wrapper generation, tool communication, trace reductions, and filters. GTI ultimately allows tool developers to focus on implementing tool functionality instead of the surrounding infrastructure. Further, we demonstrate that GTI supports scalable tool development through a lost message detector and a phase profiler. The former provides a more scalable implementation of important base functionality for MPI correctness checking, while the latter tool demonstrates that GTI can serve as the basis of further types of tools. Experiments with up to 2048 cores show that GTI's scalability features apply to both tools.**

***Keywords*-Tools, Tool infrastructure, Message Passing Interface, Scalability, Runtime error detection**

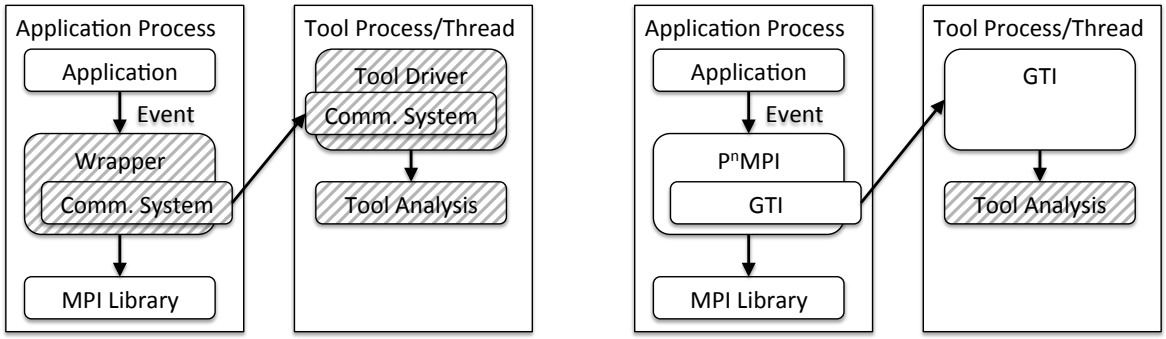
## I. Introduction

The Message Passing Interface (MPI) [1] is complex, which makes erroneous use common. Due to the richness of available MPI implementations, some errors may not manifest until a different MPI implementation or computing system is used. Runtime error detection tools can detect many of those errors, and thus, drastically reduce debugging time and increase the reliability of parallel programs. A variety of such tools exist, e.g., ISP [2], MPI-Check [3], Umpire [4], and Marmot [5].

A striking issue with these tools is that none of them provides satisfactory support for the detection of non-local errors – checks that require information from more than one process – while scaling to more than about one thousand processes. Further, each tool provides a different overall set of checks so that complete coverage would require using multiple tools. From the perspective of application developers, the situation is inefficient and does not cover errors that only manifest at scale. With petascale systems, providing test cases of less than one thousand processes may not represent the behavior of the production input sets. Thus, we are developing a new MPI runtime error detection tool called MUST [6], which combines our experiences gathered in the development of Umpire and Marmot. Our main goal with this project is the development of a scalable tool that can detect wide ranges of usage errors.

Our effort requires a scalable and flexible infrastructure to collect events, to direct them, and to facilitate their analysis efficiently. However, such an infrastructure currently does not exist. Both Marmot and Umpire use a client server approach to implement non-local checks like type matching and deadlock detection. This approach limits the scalability of the tool dramatically. Further, the placement of checks in these tools cannot be varied. They are implemented on either the application processes or on the manager/server and cannot be placed on the other, as they strictly assume where information for the checks is available. More flexibility for these correctness tools in particular, and reduced time to solution for developing scalable tools that operate on event traces in general, requires a comprehensive infrastructure. Event-based tools are all types of tools that analyze data that is intercepted during events, such as function or API calls. We present an approach for this infrastructure, the *Generic Tool Infrastructure* (GTI). It provides scalability and makes tool development more productive. We achieve this by providing a high level abstraction that focuses on the actual tasks of the tool, instead of distracting the developer with the implementation of a wide range of standard components that most tools require. Our contributions include:

- The design of a reusable, modular and scalable infrastructure for event-based tools;
- A novel abstraction to ease the development of new event-based tools;
- Integration of event aggregation on Tree Based Overlay Networks (TBONs) into the abstraction;
- Automatic generation of wrappers, a communication system, trace records, record routing, filtering, and other advanced features;




(a) Tool without GTI. (b) Tool with GTI.

Figure 1. Component illustration of MPI tools with and without GTI.

- Flexible adaption to individual use-cases and platforms;
- Two case studies of GTI-based tools, a lost message detector and a phase profiler;
- Experimental results with two widely used benchmarks for up to 2048 tasks.

Overall our results demonstrate that GTI incurs low overhead for non-local analyses and provides scalability features that apply to different tool types. Our lost message detector provides a more scalable runtime MPI message matching than any previous MPI runtime error detection tool.

Section II presents an overview of GTI and its basic principles, while Section III introduces the abstractions of GTI in more detail and illustrates them with an example. Afterwards, we present the scalability extensions that GTI provides and how it incorporates them into its abstraction (Section IV). We discuss GTI's code generation components in Section V and then introduce the lost message detector and the phase profiler in Section VI. Section VII presents our experimental results for these examples.

## II. GTI Overview

Parallel event-based tools often require significant infrastructure development, starting with the creation of wrappers that intercept events. If the tool does not analyze events locally within the application processes, it also requires the creation of trace records that capture the intercepted information, as well as a communication system that forwards these records to extra tool processes or threads that the tool spawns in order to offload analyses from the critical path. The tool implementer must develop this infrastructure before the analyses that perform event processing can be realized.

Figure 1(a) illustrates components that a tool requires to offload MPI analyses. The tool developer must provide the components that we illustrate as striped. This process for creating portable and scalable tools is time consuming, since most effort goes into the development of the infrastructure, instead of the actual tool actions, i.e., its analyses.

GTI removes this burden. The tool developer only writes the tool analyses that GTI loads, manages, and activates. Figure 1(b) illustrates the GTI tool development process. GTI handles all infrastructure related tasks, which requires the tool developer to specify which events should trigger the analyses and the overall tool layout. GTI provides the underlying infrastructure that reads this information and generates all required code including code for wrapping, trace record creation, data transport, and analysis management.

### A. The GTI Abstraction

GTI-based tool development starts with a high level abstraction that describes the tool analyses to perform. This abstraction follows a design with these properties:

- A tool consists of a set of *analyses*, e.g., for runtime error detection, analyses would be correctness checks;
- Modules contain multiple analyses (e.g., a communicator checking module can contain multiple correctness checks on communicators) and provide a C++ interface, compiled in dynamically loadable modules;
- Each module is agnostic to its place of execution, i.e., it can execute on an application process or on any other process/thread that the tool uses;
- Each module may have dependencies on other modules, i.e., a check to validate communicators can depend on a module that tracks user-defined communicators;
- XML specifications describe the inputs for each analysis within any module and which events provide them.

The developer implements the individual tool actions (analyses) as shared libraries (modules). These and existing modules cooperate to implement the overall tool functionality. The tool developer describes which data of what events these analyses require, instead of directly implementing wrappers, trace record creation, and communication. We use XML specifications for this task. A tool developer provides them with the implementation of his tool modules. Each analysis may execute locally (within the application) or on an extra tool place (thread/process). Since each module describes its dependencies, GTI can guarantee that other required modules are present at their place of execution.

### B. Software Stack

The implementation of GTI abstractions uses $P^n$MPI [7] as a base infrastructure. $P^n$MPI extends the MPI profiling

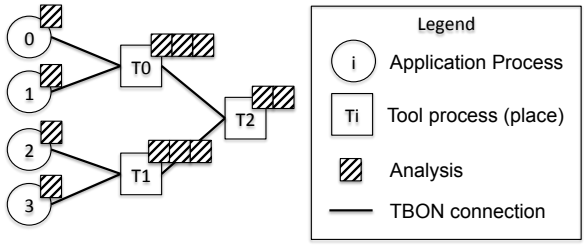



Figure 2. High-level illustration of a GTI-based tool.

interface through a tool stack abstraction so that multiple tools can be used together. More importantly, $P^n$MPI allows these tools to cooperate with each other through a service mechanism. We refer to each tool that is loaded into $P^n$MPI as a *module*. Figure 1(b) illustrates the use of $P^n$MPI as a base service for intercepting events.

For scalability, we use extra processes to which the tool can offload computations. A Tree-Based Overlay Network (TBON) connects these extra processes, or *places*.

Figure 2 shows a high level example of a GTI-based tool with four application processes and three tool places. The application processes always form the first TBON layer. Places *T0* and *T1* form the second layer, while *T2* forms a third. The example configuration uses one module (depicted as a small square) on each application-layer process, three modules on the second layer, and two modules on the third. Each module can contain multiple analyses.

GTI provides flexibility for the communication system through a mechanism that allows the tool specification to select the communication medium (e.g., TCP or MPI) and to modify communication timing and aggregation [6]. GTI also allows tool developers to use multiple TBON's in one run and trees may also lack a root, if no global analysis is necessary. Tailored communication mechanisms enable offline or trace-based tool workflows. A particular module can write events into a file for later analysis. A GTI-based tool can analyze these traces after the application run.

## III. Tool Specification

GTI provides a generator component that instantiates tools. An instance of a GTI-based tool is formed by including the $P^n$MPI library into the application, either by relinking with $P^n$MPI or by using *LD_PRELOAD*. $P^n$MPI then loads all GTI modules and activates the tool. GTI has three types of modules: ones that the tool developer provides to implement the analysis routines; intermediate modules that GTI generates automatically; GTI core modules that provide communication and offloading of tool computations. GTI generates the intermediate modules, which implement wrappers, trace records, and record forwarding and then triggers the execution of analyses. We detail the workflow that drives the generation of these intermediate modules in Section V. This section focuses on the steps that a tool developer uses to create a GTI-based tool.

### A. Component Overview

A tool instance is formed by providing three types of specifications (*layout*, *API*, *analysis*) and an installation of the tool. A tool installation consists of a set of modules and the three specifications. GTI uses a fourth specification to describe basic communication services. The individual specifications have the following tasks:

- **Analysis specification:** describes the analyses (actions) that the tool uses to provide its service.
- **API specification:** describes the functions (events) to intercept and their arguments that specific analyses use.
- **GTI specification:** describes available communication modules and drivers for tool places.
- **Layout specification:** describes the layout of a tool instance, i.e., number and size of TBON layers, selection of communication modules, and placement of analyses.

These specifications are written in XML. Besides these specifications, the tool modules consist of an implementation and a C++ interface. GTI uses the interface to invoke the module whenever an event occurs to which the API specification subscribes the module. The interface also allows modules to cooperate and provide services to each other.

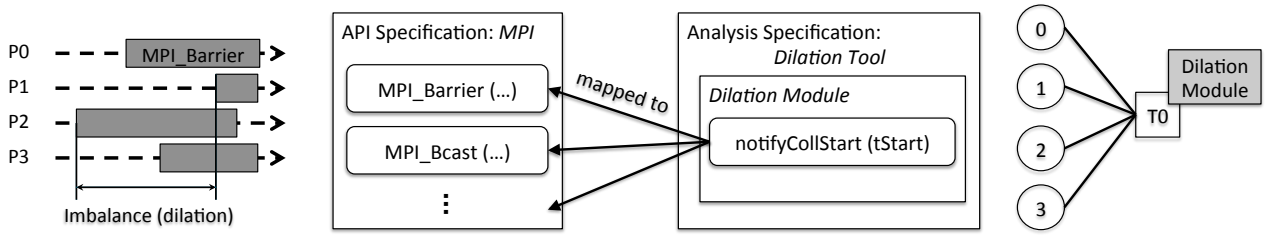
A tool developer who uses GTI provides the implementation of the analysis modules, their interfaces and their specification. The developer also specifies the events that the analyses monitor. GTI provides specification templates and developer tools to aid in this process. Finally, the tool user or developer provides a layout to instantiate the tool. Often tool developers can provide a default layout, such that users can directly use a GTI-based tool like any other tool.

### B. Example

To illustrate the GTI tool specification steps, we use a simple example tool that detects imbalances in MPI collective communication. A common source of performance problems is imbalance that causes some tasks to wait for others. One such scenario involves collective calls for which some tasks issue the collective later than others, as Figure 3(a) illustrates for `MPI_Barrier` calls. We sketch how a GTI based tool could detect such imbalances in the following.

As a first step, a tool developer must determine the information that this particular analysis requires: the imbalance detector must be informed of the start times of all collective calls to perform its analysis. It then must match collectives in order to determine when the first and last process called one of the matching collectives (this description ignores communicators for simplicity).

With GTI, the tool developer implements the collective matching and start time analysis as a module, i.e., a shared library, and provides its interface. We refer to this module as the "dilation module". To handle communicators correctly, the developer could utilize existing modules, e.g., a module for tracking communicator creation and deletion. GTI will




(a) Collective imbalance. (b) Illustration of a collective imbalance detector tool based on GTI. (c) Possible instantiation.

Figure 3. Illustration of a GTI based imbalance detector tool.

automatically notify the dilation module, whenever a process issues a collective call. We ensure this notification through the analysis and API specifications, which tell GTI that the dilation module should be informed about all collective calls.

Figure 3(b) illustrates the contents of these specifications. The analysis specification describes the dilation module and its interface, which includes a function, *notifyCollStart (tStart)*, notifies the module when a collective call is issued (collective matching may require additional inputs). The API specification describes the MPI calls – GTI provides ready to use XML specifications for MPI – and the mapping of any tool analysis to them. In our example, the specification must include all MPI collective calls, e.g., `MPI_Barrier`. It also must specify that the mapping of the dilation module's function *notifyCollStart* to these calls. A mapping can directly forward arguments of the API events to the module. However, the module in our example is interested in a start time of the collective instead of any MPI call argument. For such cases GTI allows the specification of *operations* that compute new inputs, e.g, issue a call to *gettimeofday* in order to derive the collective's start time.

Finally, the tool user provides a layout specification to instantiate the tool. Tool developers can provide scripts with default layouts to hide this complexity, e.g., MUST provides a "mustrun" script that replaces "mpirun" and handles tool instantiation. For our example, the user might specify that the dilation module runs on a single extra process as centralized collective matching requires (Figure 3(c)). The next section illustrates a scalable version of this tool.

GTI provides the user with a higher-level abstraction that generalizes the event-condition-action paradigm for event-based tools. All preceding approaches to tool infrastructure provided far more basic services such as wrapper generation. With GTI, the tool developer only implements the tool computations and describes the events that trigger them. A manual approach to the example tool would require an MPI wrapper for all MPI collectives and would have to spawn and to drive the extra tool process that runs the collective matching and time analysis, and to provide a communication medium along with suitable records for communication.

## IV. Scalability Features

A key goal of GTI is to support scalable tools while hiding as much complexity from the user as possible. In order to achieve this goal, GTI enables tools to execute analysis routines asynchronously outside of the application. GTI uses a Tree Based Overlay Network (TBON) to distribute tool analyses and provides three main features for scalable tools:

- Minimal forwarding of event records;
- Event filtering;
- Event aggregation.

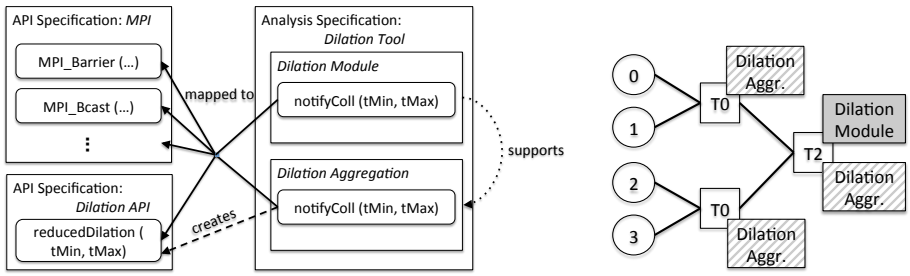
None of these features is novel by itself. However, GTI hides their complexities from the user by providing them within its abstractions. For example, MRNet [8] provides TBON functionality with filtering and event aggregation, but tool developers must manually use these features to drive scalable tool analyses. GTI includes these features within its event-condition-action paradigm. While mapping actions – executing analyses – to events is common, our abstraction supports mapping analyses to parallel events. Our scalability features within this abstraction allow tools to derive a global view from these events, while staying within the abstraction. To our knowledge, no other such approach exists.

This section first characterize how GTI achieves minimal event forwarding and filtering. We then introduce how to insert new events within our abstractions. GTI uses this concept to include event aggregation into its abstraction. We illustrate this feature with the imbalance example tool.

### A. Minimal Records and Filters

GTI automatically reduces the trace records that it forwards from one GTI place to another. It drops records and individual trace record fields that no further analysis needs. The GTI generation system (next section) enables this trace record reduction since it generates all source code that creates, receives, and forwards any event. This reduction decreases the size and number of event records that are propagated through the communication tree.

GTI's filtering functionality removes redundant or superfluous trace records from the event stream. These filters can perform condition-based filtering of events. One of our demonstration tools in Section VI uses a filter. Filters can further reduce the number of event records in the system to decrease the overall load. Filters are added as a special type of module within the GTI abstraction. Whenever these modules receive an event, they notify the system of whether or not this event can be removed through a special return




(a) Illustration of a scalable collective imbalance detector. (b) Possible instantiation.

Figure 4. Illustration of a scalable GTI based imbalance detector tool.

value for the analyses of a filter module. We only remove an event if all other analyses that would receive this event specify that they accept the filter module. Thus, no crucial events are removed for modules that are not aware of a filter. Again, these decisions are made during code generation.

### B. Injection of Trace Records

Many tools must inject additional events into the application event stream. For example, for runtime error detection we want to add events for correctness messages during the MPI event analysis. Further, if any module is only interested in an event if it fulfills a certain condition, we could inject a new event with a second module only if that condition is met. Thus, the first module does not receive any uninteresting events. Finally, we use the event injection mechanism for adding aggregations to the GTI abstraction.

GTI exposes event injection through special API events that tool specifications include. A tool developer can create a new event by adding an API call for the event to the tool's API specification. Further, such API events are marked as "wrap-everywhere" which notifies GTI that this event can be created somewhere other than the application processes (e.g., on any layer of the TBON). As the API specification defines these events, the tool developer can directly map any module to these events. GTI creates a wrapper for these events on each layer of the TBON and creates an event record when the wrapper intercepts a call. Thus, if a tool module wants to create a new event, it queries GTI for the wrapper function to use for this event and calls it. Afterwards, GTI transparently handles the creation and forwarding of an event record to any interested module. Though simplistic, this concept is extremely powerful and flexible while causing only minor extensions to the basic abstraction of GTI.

### C. Event Aggregation

Finally, we can aggregate many types of records across multiple processes, e.g., matching events of collective communication calls. Event aggregation is crucial to keep the number of events that need to be forwarded to the TBON root constant as scale increases.

GTI uses a special type of modules for event aggregations that are flagged as "aggregation modules". An aggregation module is specified and mapped like regular modules. However, like filter modules, they use return values to control the aggregation system. These values notify the driver that operates TBON nodes if a module starts an aggregation and needs further input to complete it, whether it completed, or whether inconsistent data prohibits it. We use a single communication channel that multiple aggregations can operate in parallel without losing event order [9].

Aggregation modules create aggregated events by injecting a new event into the system. They use the wrap-everywhere mechanism described above to do so. GTI removes all input events that a successful aggregation uses, and with that no redundant events occur. Other modules must specify which aggregations they support so GTI can determine when to forward aggregated events. This wrap-everywhere mechanism allows us to automate placement of event aggregations on all TBON layers to which they can be applied. This automation reduces tool development effort. In summary, we provide event aggregation within our abstraction by combining special aggregation modules and the wrap-everywhere mechanism.

We revisit the imbalance detector to illustrate event aggregation. Our first GTI-based tool used a single TBON process to run the collective matching and time analysis, which limits scalability. We must distribute the matching of the collective calls across the TBON to overcome this limitation.

As Figure 3(a) shows, the tool calculates the time dilation between the first and last process that call the collective. For distributed collective matching, we store the minimal (tMin) and maximal (tMax) start time of a partially matched collective. Both times are equal for the initial events that contain information of a single collective call. Thus, we change *notifyCollStart(tStart)* to *notifyColl(tMin,tMax)*, as Figure 4(a) illustrates. We still map the dilation module to all collective calls. We must add a new event, *reducedDilation(tMin,tMax)*, for partially matched collectives. We add a specification of the "Dilation API" with the event. To match individual collective calls correctly and to handle partially matched collectives, we must pass additional arguments to the dilation module. The concept of channel IDs [9] simplifies this process. Finally, we must add the aggregation module, which we call "Dilation Aggregation". We map its

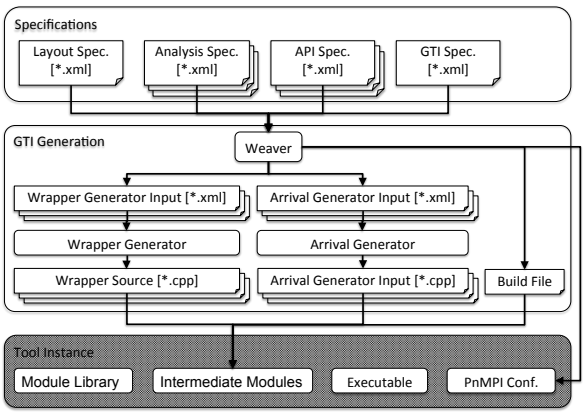



Figure 5. Overview of the GTI generation process.

analysis function *notifyColl(tMin,tMax)* to the same events as the dilation module. The aggregation module matches individual collective calls and partially matched collectives and creates its aggregated events with the *reducedDilation* API call. The dilation module also must specify that it supports the dilation aggregation.

Thus, we have a scalable version of the imbalance detector. Figure 4(b) shows a potential instantiation of the tool. It uses four application processes and two layers of additional tool processes. The aggregation module runs on both tool layers, whiler the actual time dilation module that computes the final result only runs on the root of the TBON. The dilation module receives completely matched collectives as long as no inconsistencies occur in the event stream. GTI achieves this by placing the aggregation module on the root as well. It passes any event in which the aggregation is interested to the aggregation first and then passes the aggregated events to the dilation module. All actions that we use to derive this scalable tool are done within the GTI abstractions and only require the implementations of the analysis modules and their specifications.

## V. Generation and Instantiation

Implementation of GTI's powerful abstraction requires its generation system, the *weaver*. The *weaver* provides overviews of tool modules, API calls, the layout, and input/output relationships of analyses and operations, as well as extensive detection of specification errors.

The *weaver* processes the specifications that describe the tool and its layout. Figure 5 provides an overview of this processing. The four types of specifications form the input as discussed in Section III. The weaver accepts one GTI specification and one layout specification but can use multiple analysis and API specifications. Using multiple APIs supports cases in which the tool uses an internal API for the wrap-everywhere mechanism, as in the example from the last section. Using multiple analysis specifications supports easy reuse of existing modules without restricting the combinations available to tool developers.

### A. Workflow

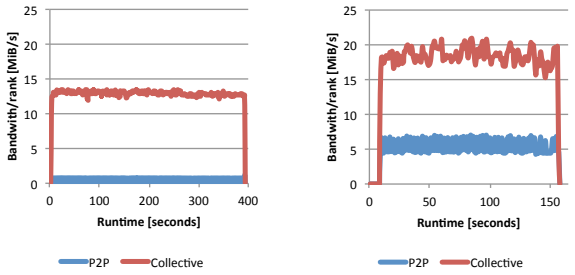
The *weaver* reads all specifications for the target tool and forms a global view from these individual elements. Rather than creating source code directly, it creates XML representations that describe the intermediate modules required to instantiate the tool. GTI has two intermediate module types: *wrapper* modules and *arrival* modules. Wrapper modules intercept API calls in the application and the tool processes (for the wrap-everywhere functionality), while the arrival modules process event records that a tool place (e.g., a TBON node) receives. GTI generates one wrapper and arrival module pair for each tool layer. Two types of code generators, the *wrapper generator* and the *arrival generator*, process the XML specifications for these modules and produce the source code for the modules. The *weaver* then generates a build file that compiles and links the sources as dynamically loadable modules.

We form the final tool instance by combining tool modules with the executable, the installation of GTI, the generated arrival/wrapper modules, and a $P^n$MPI configuration file, which the *weaver* generates. This $P^n$MPI configuration file contains all information required to connect the modules, including a list of modules for each tool layer, their linkages and the number of module instances.

### B. Processing in the Weaver

As it reads the specifications, the weaver builds a model of the overall tool. This model supports iteration over all tool layers, their individual analysis modules, and the API calls to which these analyses are mapped. A central part of the generation process calculates event record routing. This calculation first determines the arguments that each layer must receive for each API call. The weaver then generates an ID for each record that carries any arguments. The ID must uniquely determine the API call that creates the record, so that each place can identify incoming records with this ID. The XML inputs for the wrapper/arrival generator list the shape and ID of each record. Further, for each TBON layer they contain information on what parts of these records must be forwarded to adjacent layers. The weaver uses this information to schedule record routing for each layer. Further, the inputs list individual actions, including analyses and operations, which the tool modules will execute.

The weaver also automatically places aggregation modules onto the TBON layers. The weaver first analyzes which modules support aggregation and then determines for each TBON layer if we can place any of these aggregations on it. The weaver cannot place an aggregation on a layer if a descendant layer runs a module that does not support it but shares at least one input event with the aggregation. We use the same logic to place filters automatically.




(a) 512 tasks. (b) 2,048 tasks.

Figure 6. Phase profiler statistics for *143.dleslie* from Section VII.

### C. Module Generation

The two source code generators create the arrival and wrapper modules. The generated code uses trace records to store and transfer information about intercepted events. GTI provides an interface to use generic trace records. The generators query this interface for record-specific code such as creating or freeing a record instance, setting/retrieving an attribute, or record serialization/de-serialization. Thus, different implementations of this interface can produce different types of trace records, which facilitates future tool integration. GTI provides a default implementation of this interface that uses a code generation component itself in order to create records of minimal size. However, GTI could also allow use of OTF records with key-value pairs [10], which would support interesting combinations of GTI with existing performance optimization tools.

At the application layer, the wrapper module directly intercepts API events, creates the respective records, and triggers local analyses. At the tool layers, the tool must use a main loop that polls the incoming communication modules [6] to receive event records. A driver module receives records and passes them to the arrival module. Usually this driver module is specific to the mechanism that spawns tool layers. The arrival module unpacks records according to their unique ID. It passes the record data to any module that is mapped to the event. Afterwards, the arrival module reduces the record size, if possible, and it forwards the record upwards in the TBON if necessary.

## VI. Case Studies

In this section we present two GTI-based tools that demonstrate GTI's applicability and flexibility. The first tool provides basic profiling information for execution phases, while the second detects lost messages of an MPI application at runtime. Both tools use the scalability features that GTI offers, while a third GTI-based tool exists that automatically detects usage errors of MPI datatypes [11]. We introduce the two example tools and their use of GTI in this section, while we present performance results in Section VII.

### A. Phase Profiler

Our first tool comes from the field of performance analysis and provides profiling type information for MPI point-to-point and collective communication. It profiles the total amount of data that is sent at runtime along with the total number of communication calls. Users can derive average communication bandwidth or average message sizes with this information. In order to provide more detailed insight, our profiler provides this data for application phases.

Our tool uses MPI collectives that span all processes to end/start phases. The tool creates an event for profile information of the last phase at each such collective. We create the profiling events directly in-situ within the application processes. Afterwards, we aggregate them within the TBON to derive profiling information for all tasks for each phase. Finally, the root of the TBON receives the phase profiles and writes them to a trace file. Figure 6 presents performance insights with the phase profiler. It shows two profiles, one for 512 tasks (Figure 6(a)) and one for 2,048 tasks (Figure 6(b)) of an application that we introduce in Section VII (*143.dleslie*). It shows that the benchmark has an initialization phase of about 5 seconds, which issues close to no communications. Further, the profiles show that the average point-to-point bandwidth increases from less than 1 MiB for 512 tasks to more than 5 MiB for 2,048 tasks. We present the minimal collective bandwidth, as the actual bandwidth depends on the MPI implementation.

We use three modules to implement this tool with GTI:

**PhaseProfiler:** Collects communication statistics and starts/ends phases;
**PhaseAggregation:** Aggregates phase profile events;
**Tracer:** Stores phase profiles in a trace file.

We map the *PhaseProfiler* module to all MPI point-to-point send calls to store the total number of bytes that each call transfers. We also map it to all MPI collective calls to determine how much data they transfer and whether they are globally synchronizing, e.g., an `MPI_Barrier` with `MPI_COMM_WORLD` as communicator. The module creates a new event for a phase profile if a collective call is synchronizing. We use the wrap-everywhere event injection mechanism that GTI offers to create a *phaseProfile* event. The *PhaseProfiler* module requires information about MPI datatypes and communicators for its correct operation; we use existing tracking modules from MUST for these tasks.

The *PhaseAggregation* module is only mapped to the *phaseProfile* event. It matches individual phase events to create aggregated phase events. During aggregation we can directly sum up the individual entries of the phase profiles, e.g., number of bytes sent with point-to-point calls.

Finally, we map the *Tracer* module to the *phaseProfile* event and specify that it supports the *PhaseAggregation* module. The *Tracer* writes information on the phase profiles in a human readable format into an output file.

| Process 0 | Process 1 |
|---|---|
| MPI_Init() | MPI_Init() |
| MPI_Recv(from:1) | MPI_Isend(to:0, &request) |
| | MPI_Wait(&request) |
| | MPI_Isend(to:0, &request) |
| | MPI_Wait(&request) |
| MPI_Finalize() | MPI_Finalize() |

Figure 7. Example of a lost message in MPI.

### B. Lost Message Detector

Lost messages are a common correctness problem of MPI applications. The application fails to post a matching receive or send for these send and receive operations. This error can occur if a send is buffered or if a non-blocking receive operation is not completed, e.g., `MPI_Wait` is not called. Lost messages may not result in an error and the MPI implementation also may not warn about their presence when calling `MPI_Finalize`. However, according to the standard, applications with such lost messages are erroneous. Depending on the implementation, they can manifest themselves as deadlocks or can lead to the unavailability of resources, while thay can also lead to unintended matches, incorrect calculations, or even wrong application results.

Figure 7 shows a lost message example. The table shows histories of MPI calls from two processes. Where process 0 receives a single message from process 1, process 1 instead sends two non-blocking messages to process 0. The second send has no matching receive. As in most MPIs, standard mode send calls are buffered, this application may complete the second call to `MPI_Wait`, even though it was not matched. Thus, the application may complete without any error for some MPI implementations, despite the usage error of the outstanding send of process 1.

The example shows that a lost message can exist even though the application did not deadlock, while at the same time there may not exist any outstanding requests. Thus, detecting lost messages requires more than tracking whether a blocked send/receive call exists or that some request was not completed. Thus, we must observe all point-to-point send and receive calls and simulate message matching to detect all lost messages. If the simulated message matching reveals any outstanding messages when the last `MPI_Finalize` call is issued, we report the lost messages to the developer.

We select this example as message matching is a major challenge for scalability from our experiences with Umpire and Marmot. These correctness tools need message matching in order to execute type matching and to model wait states for deadlock detection. We present two GTI based implementations, a first implementation that performs a centralized message matching running on the root of the communication tree and a distributed message matching that runs on the entire communication tree.

*1) Centralized Message Matching:* The centralized implementation uses three key analysis modules:

**LostMessage:** Performs the message matching;
**WcUpdate:** Monitors for completed wildcard receives;
**CreateMessage:** Logs lost messages.

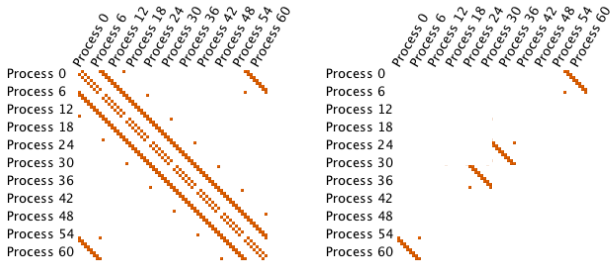
The *LostMesage* module manages lists of outstanding messages. It uses special handling for wildcard receives, which are kept in an extra list of outstanding wildcard receives until a potential match is found. Once we find a potential match, the message matching simulation cannot simply use the first match that it detects, but rather must use the match that the MPI implementation decides. We update the source of the wildcard receive to ensure the correct match. The *WcUpdate* module retrieves this information from the status that the application retunrs to the `MPI_Recv` call, or the respective completion call if the receive was initiated with `MPI_Irecv`. The module uses a wrap-everywhere event to forward this information. It only creates the event when a wildcard receive completes, in order to keep the number of events in the system as low as possible.

The *LostMesage* module needs information on persistent requests, and communicators in order to operate correctly. It also uses the respective tracking modules from MUST. Finally, when the lost message tool detects that an unmatched send or receive call still exists, the implementation supports the usage of the *CreateMessage* module. This module is part of MUST and forwards log events to a logger that may write the events in a sorted and easily readable format like HTML.

*2) Distributed Message Matching:* The centralized implementation of the lost message tool is not scalable, since the number of messages that are analyzed at the root of the TBON increases with scale. We use a distributed message matching implementation to overcome this limitation. Each node in our TBON receives point-to-point send/receive events from a set of processes. Each node can match all sends for which the destination argument is a rank within this set, and all receives for which the source argument is a rank within this set. All sends/receives that can be processed on a certain node do not need to be processed by any further node, while sends/receives that cannot be matched by the current node must be forwarded to the next layer of the communication tree. Finally, the root can process all remaining sends/receives.

Our distributed matching reuses the *LostMessage* module. We augment it with a filter module. We now run the *LostMessage* module on all TBON layers to distribute the message matching. We map the filter module to all point-to-point events and to check whether we can process a send/receive on the TBON current node. If so, the event is filtered out and is analyzed by the *LostMessage* module running on the current node, otherwise it is forwarded to the next layer in the tree.

This strategy has two limitations: the processing of wildcard receives; and an inefficiency of TBONs for message matching. The first limitation results from the nature of wildcard receives. When a TBON node discovers a wildcard receive, which process will match it is unclear. Thus, it




(a) Communication pattern of NAS-NPB *bt*.

(b) Parts of the pattern of NAS-NPB *bt* matched on the root.

Figure 8. Visualization of the parts of a communication pattern that need to be matched on the root for 64 processes.

is unclear which TBON node should process the receive. Forwarding it to the root of the tree is not sufficient since we would also have to forward its matching send to the root, although that send is likely to be processed and filtered out by another node. Our current distributed implementation therefore does not support wildcard receives and aborts if one is observed. A possible solution would use the following handling strategy: when a TBON node discovers a wildcard receive, it waits until the arrival of the updated source, which the *WcUpdate* module provides. If it arrives, we can turn the wildcard receive into a regular receive and process it as such. If it does not arrive, than the call was either part of a deadlock, or it was a non-blocking receive for which no completion call was issued. Both cases represent erroneous MPI usage by themselves and can be reported to the user. However, waiting for the update requires that we buffer all events that arrive after the wildcard receive and that processing continues only after we receive the updated source, which may be expensive. We do not test this extension, as our main intention was to evaluate whether such a distributed message matching is suitable for correctness checking purposes at scale.

The second limitation results from an inefficiency of tree networks for message matching. Consider a scenario with a TBON root being connected to two child nodes. Each node provides sends/receives from about $50\%$ of the processes and the root must match all messages between these two sets of processes, which is $50\%$ of all existing send-recv pairs.

Figure 8(a) shows a communication pattern of the *bt* kernel of the NAS parallel benchmarks, as visualized by many performance tools. In this representation, a cell with coordinates $(x, y)$ is filled if process $x$ sends a message to process $y$. Figure 8(b) shows which communication partners would be matched on the root for the pattern shown in (a). More generally, the layers closer to the root have more send-receive pairs to match than the prior TBON layers, which shifts large parts of the message matching workload onto processes close to the root. However, this approximation only holds for scenarios in which all processes communicate with all other processes or with an arbitrary communication pattern. Most applications use far more regular patterns in which each process only communicates with a small set of other processes, often neighbors. Figure 8(a) is an example for such an application. The results in the next section show that this limitation may only have a small impact in practice.

### C. Discussion

The modules of the phase profile example are written in about 700 lines of code (LOC)[1], while the centralized lost message detector uses about 1,200 LOC. Both tools use further tracker modules from MUST that require about 3,000 LOC in total. However, GTI generates about 17,000 LOC for MPI wrapping, record routing, trace records, and record arrival while 5,000 additional LOC of GTI modules are used in our tool instantiations. Thus, the tool developer only needs to provide a small fraction of the source code that constitutes the tool, thus decreasing the time to solution besides the increased flexibility resulting from GTIs exchangeable communication and offloading system.

Also, GTI provides scalability features for both tools within its basic abstraction. For the lost message detector, the additional filter module that allows distributed message matching is just about 250 LOC. Finally, with GTI tool modules, we can easily exchange components. For both of our example tools, we reuse existing MUST modules. Interestingly, even though our phase profiler is a performance tool, we could use components of a correctness tool without any modification. Thus, wide usage of GTI could simplify cooperation for future tool development dramatically.

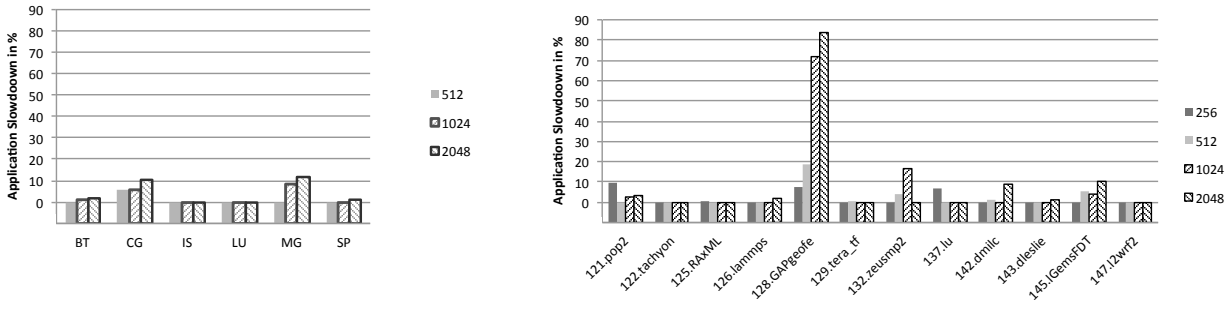
## VII. Application Results

This section presents performance results for the tools from the previous section. We use an 864 node Opteron Linux cluster with a QDR InfiniBand network for all experiments. Each node has 16 cores on four sockets and 32 GB of main memory that is shared between all cores. As benchmarks we use the NAS Parallel Benchmarks (NPB) [12] version 3.3 and SPEC MPI2007 [13] version 2.0. For NPB we use problem size E for the phase profiler and problem size D for the lost message detector since it has higher overheads. For SPEC MPI2007 we use the *lref* data set. While NPB presents rather simple codes, SPEC MPI2007 presents more complex applications that are derived from production codes. We present results for the phase profiler first, followed by results for the centralized detection, and finally results for its the distributed version.

### A. Phase Profiler

We instantiate the phase profiler with a layout where each TBON node receives events from at most 32 processes. For example, we use 32 nodes on the first TBON tool layer for 1,024 processes and one additional node on a second layer. We use MPI-based communication with an aggregation system to derive a highly optimized communication medium

[1]Without headers, including comments and empty lines.




(a) NPB.

(b) SPEC MPI2007.

Figure 9. Slowdown of the phase profiler.

within the TBON [6]. Figure 9(a) shows the slowdown of the phase profiler in percent for NPB, whereas Figure 9(b) shows the slowdown for SPEC MPI2007. As the kernels *bt* and *sp* only support a square number of processes, we use 529 tasks instead of 512 tasks and 2,025 tasks instead of 2,048 tasks for them. For simplicity we use the approximate powers of two in our performance charts. We excluded the benchmarks *ep* and *ft* since they do not use point-to-point calls and are thus uninteresting for our lost message detector. We also omit *is* since it is not available for class E.

The overhead is about 10% or less for all tests, except for *128.GAPgeofem*. This benchmarks issues about 1,500 globally synchronizing collectives per second and tasks, for the run with 1,024 processes. We hypothesized that this saturated TBON bandwidth, as each node must serve up to 32 tasks. However, experiments with more TBON nodes do not improve performance. We assume that the extra communication on the application processes causes the slowdown, as this benchmark is already communication bound at 1,024 tasks.

### B. Centralized Detector

We use a single additional tool process to instantiate the centralized lost message detector and use our MPI-based communication system. Figure 10(a) shows overheads for the NPB with up to 1,024 tasks and Figure 10(c) shows overheads for SPEC MPI2007 with up to 2,048 tasks. We abort some SPEC MPI2007 tests since their overhead was above 100%. We measure high overheads for the kernels *lu* and *121.pop2*[2] that result from their large numbers of point-to-point messages. At 1,024 tasks, the tool causes an unacceptable slowdown of more than 100% for most benchmarks due to the increased workload on the single process that runs the detector. When doubling the scale, the total number of sends and receives doubles or triples in many cases while the base runtime decreases.

### C. Distributed Detector

Our tool instantiation uses one TBON node per 32 tasks while the remaining layers form a binary tree for the distributed version of the lost message detector. As in the previous experiments we use MPI-based communication. Figure 10(b) presents the performance results for NPB with the distributed detector, while Figure 10(d) shows the results for SPEC MPI2007. Besides the benchmarks that use no point-to-point communication (*ep*, *ft*, *125.RaxML*) we excluded *is* as the results for the centralized case scale well. Further, we exclude *lu*, *137.lu*, and *142.dmilc* since these benchmarks use wildcard receives that the current distributed implementation does not support.

The results show that the distributed implementation significantly reduces overhead. The only benchmark that causes more than 100% slowdown is *mg*, which has a runtime of only 5 seconds with 2,048 tasks, so tool initialization and shutdown become significant. It reduces the overhead for some benchmarks by an order of magnitude.

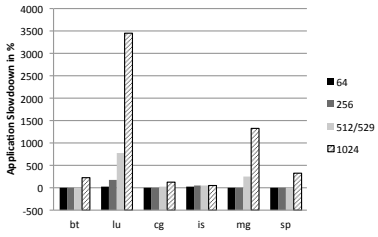
[2]We use the ltrain input for *121.pop2* for the lost message detector.

## VIII. Related Work

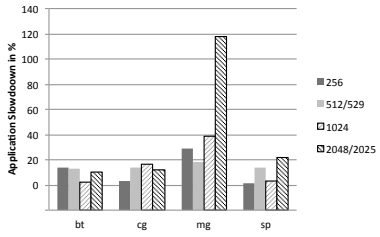
First, this work relates to parallel or MPI specific tool infrastructures, such as DeWiz [14], STCI [15], and $P^n$MPI [7]. While this work is based on $P^n$MPI, DeWiz is specific to analyses in event graphs that use a specific protocol. STCI is a fine grained and low-level approach that is designed to provide minimal communication overheads. None of these approaches provides a high level abstraction or a powerful generation component such as GTI.

GTI also relates to TBON communication work, including MRNet [8], which is widely used on many systems. GTI's strengths are its abstraction and its generation component, The communication system is flexible and integration or usage of MRNet is future work. Open questions include the compatibility of MRNet and $P^n$MPI modules, as well as the relationship of the GTI aggregation modules and filters to MRNet streams and filters. Integration would require that we adapt our order preserving event aggregation system [9] into MRNet. Further, tools that use their own TBON implementations include DDT [16] and Periscope [17] We could adopt optimizations in their communication systems into GTI's communication modules.

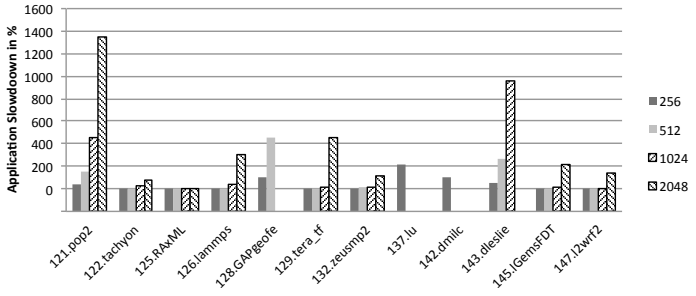
Finally, GTI's design and our lost message detector relate to approaches to runtime MPI error detection, including ISP [2], Marmot [5], Umpire [4], and MPI-Check [3].



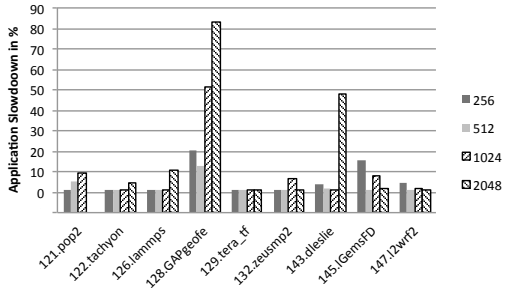

(a) NPB, centralized detection.




(b) NPB, distributed detection.




(c) SPEC MPI2007, centralized detection.




(d) SPEC MPI2007, distributed detection.

Figure 10. Slowdown of the lost message detector.

## IX. Conclusions

We presented GTI, a scalable tool infrastructure for tools in parallel systems. Our main usage scenario currently is runtime MPI error detection and was motivated by the need for more efficient, flexible, and maintainable tool development. In contrast to existing approaches, GTI offers a high level abstraction that is well suited to the specification of scalable tools. The abstraction is backed by a generation-based engine that automatically handles many common and time consuming steps in tool development. Its capabilities include the generation of wrapper and routing modules, of trace records, a communication system, and the offloading of tool analyses. Scalability features in GTI support tools that benefit from its TBON-based communication system.

Two GTI-based demonstration tools implement a lost message detector for MPI and a phase profiler. Both tools can naturally use GTI's scalability features and incur low overheads at 2,048 tasks. The lost message detector implements MPI message matching that is a severe scalability limit of our previous approaches to MPI runtime checking. We used a first distributed MPI message matching running in a TBON. It provides performance improvements of an order of magnitude. However, our distributed matching may cause load imbalances for different communication patterns or higher scales. We have two options to overcome these limitations. First, we could dynamically reconfigure the tree in order to adapt connections such that the workload is balanced within the tree. Second, we could use a special tool layer that provides intracommunication within the layer, in order to compute message matching in a single tool layer. In this case nodes could forward sends/receives that they cannot process to other nodes within the same layer.

Our experience with the two demonstration tools and our MUST prototype shows that the GTI abstractions are natural and support efficient development of non-trivial tools. We reused many modules of the MUST prototype without modification in our tools. Thus, GTI is a promising platform for sharing tool components within the community.

## Acknowledgments

We thank the ASC Tri-Labs and the Los Alamos National Laboratory for their friendly support. Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-CONF-522031).

## References

[1] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard, Version 2.2," http://www.mpi-forum.org/docs/mpi22-report.pdf, April 2009.

[2] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby, "ISP: A Tool for Model Checking MPI Programs," in *PPOPP*, 2008, pp. 285–286.

[3] G. R. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou, "MPI-CHECK: A Tool for Checking Fortran 90 MPI Programs," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 2, pp. 93–100, 2003.

[4] J. S. Vetter and B. R. de Supinski, "Dynamic Software Testing of MPI Applications with Umpire," *Supercomputing, ACM/IEEE 2000 Conference*, pp. 51–51, 04-10 Nov. 2000.

[5] B. Krammer and M. S. Müller, "MPI Application Development with MARMOT," in *PARCO*, ser. John von Neumann Institute for Computing Series, vol. 33. Central Institute for Applied Mathematics, Jülich, Germany, 2005, pp. 893–900.

[6] T. Hilbrich, M. Schulz, B. R. de Supinski, and M. S. Müller, "MUST: A Scalable Approach to Runtime Error Detection in MPI Programs," in *Tools for High Performance Computing: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds., ZIH. Springer Publishing Company, Incorporated, 2009.

[7] M. Schulz and B. R. de Supinski, "$P^n$MPI tools: A Whole Lot Greater than the Sum of Their Parts," in *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, Nov. 2007, pp. 1–10.

[8] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, ser. SC '03. New York, NY, USA: ACM, 2003, pp. 21–.

[9] T. Hilbrich, M. S. Müller, M. Schulz, and B. R. de Supinski, "Order Preserving Event Aggregation in TBONs," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2011, vol. 6960, pp. 19–28.

[10] A. Knüpfer, M. Geimer, J. Spazier, J. Schuchart, M. Wagner, D. Eschweiler, and M. S. Müller, "A Generic Attribute Extension to OTF and its Use for MPI Replay," *Procedia Computer Science*, vol. 1, no. 1, pp. 2109–2118, May 2010, proc. of the International Conference on Computational Science (ICCS).

[11] J. Protze, T. Hilbrich, A. Knüpfer, B. R. de Supinski, and M. S. Müller, "Holistic Debugging of MPI Derived Datatypes," in *IPDPS 2012: Procedings of the 26th IEEE International Parallel & Distributed Processing Symposium*, 2012.

[12] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon, "NAS Parallel Benchmark Results," IEEE Parallel and Distributed Technology, Tech. Rep., 1992.

[13] "SPEC MPI2007 Benchmark Suite for MPI," http://www.spec.org/mpi2007/.

[14] H. Brunst, D. Kranzlmüller, and W. E. Nagel, "Tools for Scalable Parallel Program Analysis - Vampir NG and DeWiz," *The International Series in Engineering and Computer Science, Distributed and Parallel Systems*, vol. 777, pp. 92–102, 2005.

[15] D. Buntinas, G. Bosilca, R. L. Graham, G. Vallée, and G. R. Watson, "A Scalable Tools Communications Infrastructure," in *Proceedings of the 2008 22nd International Symposium on High Performance Computing Systems and Applications*, ser. HPCS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 33–39.

[16] D. Lecomber, "Debugging the Future with DDT at ORNL," http://www.nccs.gov/wp-content/uploads/2009/06/DDT_ORNL_Tech_Day_1109.pdf, Apr. 2011.

[17] M. Gerndt, K. Fürlinger, and E. Kereku, "Periscope: Advanced Techniques for Performance Analysis," in *PARCO*, ser. John von Neumann Institute for Computing Series, vol. 33. Central Institute for Applied Mathematics, Jülich, Germany, 2005, pp. 15–26.